Self-stabilizing Load Distribution for Replicated Servers on a Per-Access Basis

Felix C. Gärtner* and Henning Pagnia
Computer Science Department
Darmstadt University of Technology
D-64283 Darmstadt, Germany
Email: {felix|pagnia}@informatik.tu-darmstadt.de

Abstract

Usually, load distribution schemes for replicated servers are based on a many-to-one mapping between client and server, meaning that while a server may serve many clients, a client has a single specific server which it queries at any point in time. In some cases, however, it is desirable that the number of accesses of a client may be distributed over multiple servers, thus yielding a many-to-many mapping between clients and servers. In this paper, we present a simple method to efficiently realize such a many-to-many mapping between clients and servers. For the sake of transparency, we add a component called "distribution module" to the communication interface of client and server. This module is responsible for distributing server accesses over multiple target machines in a well defined way. We present algorithms for the client and server component and show that they are self-stabilizing, meaning that they converge to a stable state once the access pattern becomes regular. Due to this property, the components can tolerate any internal transient fault in a non-masking way. Additionally, our approach is highly modular since servers may run an off-the-shelf load distribution algorithm and replica consistency is not affected.

1. Introduction

In large distributed systems like the Internet, the mere number of clients that may want to access a particular network service surmounts any attempt to implement it on a single machine. One possible solution to this problem is to replicate the servers and to have clients access their "nearest" server whenever they require to be served. Such a system can be enhanced by *load distribution*, i. e., by letting clients favor lightly loaded machines leading to a more equally distributed load among all servers. There is a diversity of different terminology in the literature; especially the terms "load balancing", "load distribution", and "job distributing" are often used inconsistently. Throughout this paper we will use the terminology of Shivaratri et al. [19] where the term *load distributing algorithm* refers to any algorithm which improves the overall system performance by transferring some form of processor load (e.g., a task or process) from heavily loaded machines to lightly loaded machines. Furthermore, we will often call load distributing algorithms *load distribution algorithms*. A *load balancing algorithm* is a load distribution algorithm which tries to equalize the loads at all computers. We are mainly interested in load balancing here, so we will use the terms load balancing, load distributing and load distribution algorithm interchangeably.

The assumption underlying many existing load balancing schemes is that load is well distributed if the number of clients associated with each server is approximately equal. An important distinction made in this context is the one between static and dynamic load balancing [19]: In *static load balancing schemes* the client has a fixed server to connect to whenever it wishes to be served. In *dynamic load balancing schemes*, this server may change dynamically according to the algorithm used. The important point is that in both cases a client has a *single* specific server to query at any given point in time. This is in contrast to server machines, which may serve *all* clients concurrently. Mathematically this can be viewed as a many-to-one mapping between clients and servers. Although this mapping may be dynamic, there exist scenarios where — due to this asymmetry — any load balancing effort is wasted. These cases arise when a single client is the main cause for high loads on a single server. In such circumstances it is desirable that all accesses of every client may be distributed over multiple servers, thus yielding a many-to-many mapping between clients and servers.

In this paper, we present a simple method to efficiently realize such a many-to-many mapping between clients and servers. Server load is measured in the number of accesses it receives within a certain period of time. Clients are not associated to a single server anymore but distribute their ac-

*This author's work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the "Graduiertenkolleg ISIA" at Darmstadt University of Technology.

cesses among a set of servers. The distribution method is based on local access statistics and client/server cooperation. It is implemented transparently by adding a "distribution module" to the communication interface of client and server. (Adding such a module can be viewed as introducing or augmenting a proxy both at the client and the server.) We give algorithms for both modules and show that they are self-stabilizing, meaning that the system behavior converges to a stable state once the access pattern becomes regular. The self-stabilization property of our modules implies a strong form of fault tolerance: they can automatically recover from any transient fault in a non-masking way. The approach is also highly modular since servers may run any off-the-shelf load distribution algorithm and replica consistency is not affected. Our experience with this scheme suggests that it can be an uncostly add-on to many existing load distribution schemes to provide a finer grained load distribution than the original solution.

This paper is organized as follows: We start with a description of the assumptions underlying our system model in Section 2. In Section 3 we then present our method and give an execution example. We prove the self-stabilization property of the scheme in Section 4. Related research is presented in Section 5. Section 6 concludes this paper and sketches directions for future work.

2. System Model

We consider a network of computing machines with an unknown but connected topology. Globally accessible services exist which grant access to data items. For the sake of fault tolerance and performance, identical copies of these services exist on multiple machines. Those machines that store a version of the replicated data are called *servers*, all others are called *clients*. We assume that all machines have unique identifiers and that they have access to local hardware clocks approximately synchronized by a distributed time service like NTP [17]. Clients and servers run a set of concurrent processes that synchronize locally by using a *notification mechanism*. Processes on different nodes may communicate though a communication subsystem that offers *synchronous message passing*. (Message passing is synchronous if there is a bounded delay on message delivery time.) Message delivery is assumed to be point-to-point and reliable with FIFO delivery order. Messages are always tagged with the identity of the sender. The latency of inter-node message passing is assumed to be much higher than intra-node notifications. For simplicity we assume that the latter is instantaneous. As we have mentioned above, we also assume that the network forms a connected graph (i.e., there exists a communication path between each pair of nodes).

Clients run a distinguished process called the *application* process that continuously generates requests to the replicated data. Servers run a special *data* process that serves application requests by retrieving the requested data item from the replicated data. In this paper, the local algorithms of the processes are presented in an event-driven notation. We assume fair event scheduling, meaning that an event which is possible for long enough will eventually be scheduled to take place.

In order to be able to prove the stability of our scheme we must define accesses and regular access patterns: A node i *accesses* an object on another node j by sending a special request message to j and subsequently receiving a reply message from j. For presentation purposes only we assume the existence of a global clock and let S_i^t denote the local state of node i at real time t. As mentioned above, a node i also has a local clock t_i that naturally may differ from the ideal global clock. A *phase* of node i is a fixed time interval of t_i and θ_i is the real time length of this time interval. A node i performs a *regular access pattern* if and only if the accesses of node i to other nodes (and thus local state transitions) occur at the same instances in time within every phase, i.e., $S_i^t = S_i^{t+\theta_i}$. Note that the regularity of an access pattern of a node i depends only on the local clock of i; it does not depend on the time of the global clock when a phase on node i begins.

3. The Basic Algorithms

This section contains brief descriptions of the algorithms that run within the distribution modules on both the client and the server side. We first present the compositional structure of our approach and show how it interacts with the load distribution algorithm and the replica consistency scheme in Section 3.1. After presenting the concrete algorithms for client and server (Sections 3.2 and 3.3) we discuss an example.

3.1. Compositional Structure

In our approach there are two types of modules added to an existing system: one on the side of the client and one at the server side.

Client side. The compositional structure of the client side is depicted in Figure 1. It shows the application on the left hand side which is the source of read requests to the replicated data stored on the servers (action 1). These accesses pass through the client side distribution module (2) which distributes the accesses among the servers according to a local algorithm (3). The results of these accesses (4) are also received by the module (5) and passed back to the application (6). However, the distribution module also receives special *server instruction* messages which it uses to guide the distribution process of subsequent requests by the application. The local algorithm of the client side distribution module as well as the nature of these server instruction messages will be explained in Section 3.2.

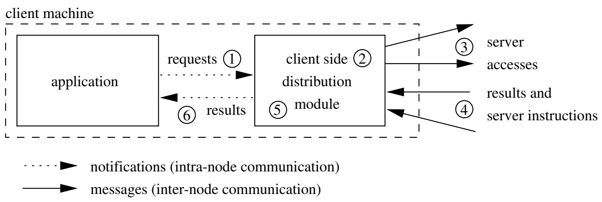

Figure 1. The compositional structure of the client.

Server side. Figure 2 shows the compositional structure of the server side. Client accesses (1) pass through the local distribution module (2) and are directed towards the replicated data stored on the server (3). The results of these requests are handed back to the distribution module (4) and are subsequently sent back to the client (5). Additionally, the distribution module counts the number of accesses and passes a local load count to a load balancing module (6). The load balancing module uses this value together with the load counts of neighboring servers to decide on whether a certain amount of load must be moved between the servers (7) to balance or distribute the load. If necessary, it issues "move" instructions (8) to the distribution module which then sends server instructions messages to some clients (9). These messages ensure that clients will shift a certain *proportion* of future requests to a different server.

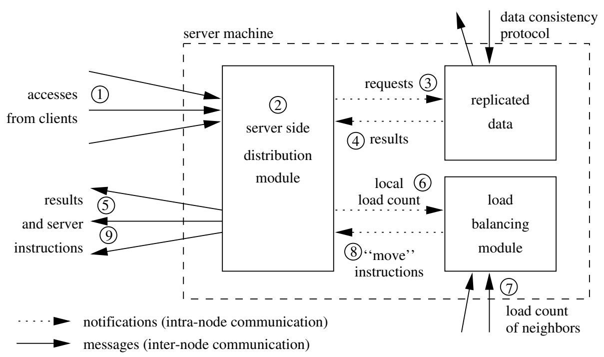

Figure 2. The compositional structure of the server.

Specification of load balancing scheme. The only restrictions we place on the particular load balancing algorithm are described by the following abstract functionality [4]: Associated with every (server) node i in the network is a variable integer value of *load units*, called the *local load count of node* i. Load units may be arbitrarily produced or consumed at any time and at any place in the network. Also, the following properties must hold:

- (Invariance) The load balancing algorithm itself does not produce or consume load units. It may merely move load units between nodes by issuing special "move" instructions (see Figure 2).
- (Convergence) Starting from any state (i.e., an arbitrary load distribution in the network) and provided that no load units are produced and consumed in a sufficiently long period of time, the program will reach a state where the loads on all nodes are equally distributed according to some stability predicate. (An example would be the predicate: "the load on all nodes differs by at most 1.")
- (Closure) Starting from a state where the stability predicate holds: if no load units are produced and consumed, the next state will also satisfy the stability predicate and the algorithm will not issue a "move" instruction.

A large body of existing load balancing schemes can be tailored to satisfy this interface. They may even be fully distributed and decentralized. Particularly suitable algorithms have been proposed by Arora and Gouda [4] and by Grønning et al. [14]. Note also that due to the use of a stability predicate this specification does not only cover load balancing algorithms but also many load distributing algorithms in the sense of Shivaratri et al. [19].

Replica consistency. Replica updates are not of interest here. We assume that replicas are kept mutually consistent by a specific replica control protocol (for example that of Golding [13]) which runs in parallel with our algorithm.

3.2. Local Algorithm on Client

The local algorithm running within the distribution module on the client side is shown in Figure 3. Its data structures consist of two tables which count the number of accesses passing through the module during a phase. The contents of T describes the access pattern of the client within the last phase, C contains accumulated access statistics from the current phase. At the end of a phase the value of C is assigned to T before C is reset (action 4).

The algorithm uses the access pattern T from the last phase to select a server for the next incoming request from the application (action 1). The selection policy is encapsulated within a function *next* which takes an access pattern and returns server ids in such a way, that the access pattern is proportionally reproduced. For example, in a scenario with three servers and an access pattern $(1, 2, 0)$ the function *next* would never return the id of the third server, and return the id of the second server twice as often as that of the first server.[1] Data returning from the server is simply relayed back to the application (action 2).

Data structures:
T **array** $[1..n_s]$ **of** $\mathbb{N}$ **init** $[0,\ldots,0]$
C **array** $[1..n_s]$ **of** $\mathbb{N}$ **init** $[0,\ldots,0]$
Actions:
⟨1: upon receiving a request notification r from the application⟩ **do**
$j :=$ *next*(T); {* get next server id *}
$C[j] := C[j] + 1$;
send r **to** j;
⟨2: upon receiving a response d from server j⟩ **do**
notify application with d;
⟨3: upon receiving a "move m to v" instruction from server j⟩ **do**
$C[j] := C[j] - m$; $C[v] := C[v] + m$;
⟨4: upon end of current phase⟩ **do**
$T := C$; $C := [0,\ldots,0]$;

Figure 3. Local algorithm of client side distribution module at client i.

If the client receives a server instruction message (action 3) which tells it to shift a certain amount m of accesses to another server v, then the client simply updates its future access pattern C to reflect this.

3.3. Local Algorithm on Server

The local algorithm of the server is shown in Figure 4. It also has a table C of current access statistics from the clients. Every time a client requests an item from the replicated dataset (action 1), the access is counted and the requested data is sent back after querying the database (action 2). The access statistics in C within a certain fixed period of time are an indication of the server's load and are sent to the load balancing module periodically before they are reset and re-established (action 4).

Now for the most interesting bit of the algorithm: from time to time the server will receive a message from the load balancing module. Such a message indicates that the load is currently high and that a certain amount of load (namely m periodic accesses) should be moved to another server (namely v, see action 3 in Figure 4).

To follow this instruction, the server splits up the requests over all clients that it recently received read requests from. This is encapsulated within the function *split* which returns a set of pairs (m_c, c) signifying that client c should shift m_c periodic accesses to the other server. Finally, all clients selected by the *split* function are informed about this action by sending appropriate "move" messages to them.

[1] The function *next* must also reproduce the same access pattern given the same value for T.

Data structures:
C **array** $[1..n_c]$ **of** $\mathbb{N}$ **init** $[0,\ldots,0]$
Actions:
⟨1: upon receiving a request r from client i⟩ **do**
$C[i] := C[i] + 1$;
notify data process of request r **from client** i;
⟨2: upon receiving a notification from data process on data d for client c⟩ **do**
send d **to** i;
⟨3: upon receiving a "move m to v" notification from load balancing module⟩ **do**
forall $(m_c, c) \in$ *split*(m, C) **do**
send "move m_c to v" **to** c;
⟨4: upon end of current phase⟩ **do**
notify load balancing module with current load $\sum_k C[k]$;
$C := [0,\ldots,0]$;

Figure 4. Local algorithm of server side distribution module at server j.

3.4. Example

Consider a scenario with two servers s_1 and s_2 and two clients c_1 and c_2. Within a certain time period client c_1 accesses server s_1 three times and c_2 accesses server s_2 once. This fact and the relevant data structures are depicted in Figure 5 (a). For the sake of simplicity, let us assume perfectly synchronized clocks meaning that the phases in all local algorithms end at the same global instance in time. Then it is quite easy to see that this state is a stable state, meaning that the depicted data structures and thus the access pattern will not change.

If the stability predicate of the load balancing module is not satisfied, sooner or later the module of, say, s_1 will witness an imbalance of load and initiate move of load. In our example it could state "move 1 to s_2" meaning that one load unit should be transfered to s_2. Now s_1 "splits" this single load unit over all clients that it serves (namely c_1) and sends move instructions to them. Thus, c_1 will subsequently receive a message from s_1 to "move 1 to s_2" and update its local access pattern C. After the next timeout, C will become valid to the *next* function and c_1 will direct one in three accesses to s_2 (see Figure 5 (b)).

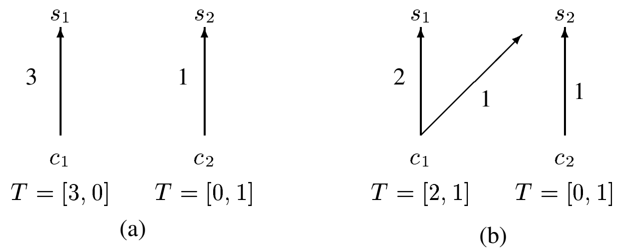

Figure 5. Example scenario (a) before and (b) after actions of the load balancing module.

4. Discussion

In this section, we show that the algorithm is self-stabilizing and we present a brief analysis of its performance.

4.1. Self-stabilization

The algorithm in question here is a composition of two distributed algorithms: the load balancing algorithm and our distribution modules. This can be exploited by composing the proof of the self-stabilization property [20]. For this we must show, that

A1 the load balancing algorithm self-stabilizes,

A2 the distribution modules self-stabilize given the stability of the load balancing algorithm, and

A3 the distribution modules do not interfere with the load balancing algorithm.

Requirement A1 is part of the specification of the load balancing scheme from Section 3.1. Property A2 is the actual self-stabilization property and will be shown after proving the "noninterference" property A3.

A3 Noninterference. From the algorithm it is obvious that every application request is transparently forwarded to a server within the distribution module on the client (action 1 of Figure 3). Within the server side distribution module, the access is served by querying the database and returning the result to the caller (actions 1 and 2 of Figure 4). Through action 2 (of Figure 3) of the client side distribution module the result eventually arrives at the application. So the distribution modules relay server accesses and replies transparently.

Noninterference with the load balancing module is not as easy to prove. The reason for this is that "move" instructions have a certain latency because such an instruction takes effect on the current access statistics C of the client and not on the previous access statistics T that govern the access distribution observed by the server. We must ensure that the amount of load "moved" by the algorithm must become effective on the target server before the load balancing algorithm re-evaluates its stability predicate. It is shown in Lemma 2 that it takes two phases for a move instruction to become effective. Thus, if the overall load balancing algorithm evaluates its stability predicate in larger intervals the noninterference property holds.

A2 Self-stabilization. Property A2 leaves us with the usual proof obligations for self-stabilization [3, 18]:

B1 (Closure) The set of stable states of the algorithm is closed under system execution.

B2 (Convergence) Given an arbitrary state, the algorithm will reach a stable state in finite time.

Stable states. The stable states of the algorithm are characterized by the fact that nothing new happens within each phase, i.e., the behavior within a phase is equal to that within the previous phase. Let $i.C^t$ denote the value of the access pattern C on node i at time t and θ be the length of a phase. Then the stable states are formally defined by the predicate

$$\begin{aligned} P \equiv & \\ & \forall i \in \mathit{Clients} : \left(i.C^t = i.C^{t-\theta} \wedge i.T^t = i.T^{t-\theta} \right) \wedge \\ & \forall j \in \mathit{Servers} : j.C^t = j.C^{t-\theta} \end{aligned}$$

In order to prove stability, we first make some additional assumptions: (1) the number of accesses generated by the application as well as their invocation time are the same within every phase, (2) messages are delivered with constant delay, and (3) clocks are perfectly synchronized. While being rather unrealistic, these assumptions simplify the proof substantially. We will return to these assumptions later and show that weakening them merely influences the *efficiency* of the scheme, not its *correctness*.

B1 Closure. To show closure, we have to prove the following Lemma.

Lemma 1 *If P holds, and the access pattern is regular, and message delivery delay is constant, then P will hold again after one round of the algorithm.*

Proof. To show closure it suffices to show that $C = T$ at the end of a round on the client side. On the server side it must be shown that $C = C'$ at the end of a round where C' denotes the value of C at the end of the last round. However, both facts are obvious from the fact that the *next* function reproduces T on the client side and the access pattern is the same within every round. If T is reproduced on all clients then every server will receive the same access statistics in C. □

B2 Convergence. Convergence is usually shown by exhibiting a termination function which decreases with every step of the algorithm and which is bounded from below. This is not very difficult here, since we have a constant stabilization time as shown by the following Lemma.

Lemma 2 *If the access pattern is regular and message delivery delays are constant, the system will reach a state in P after at most two rounds.*

Proof. Let c be an arbitrary client and let the values of $c.T$ and $c.C$ be arbitrary. Without loss of generality let us assume that a new round has just begun, which means that $c.C$ has been reset (action 4 of Figure 3). In this new round, the client — using the *next* function — will distribute its accesses over the servers according the value of $c.T$. These accesses will be counted in $c.C$ and will reflect the access pattern of the client within a round. At the beginning of the next round, $c.T$ will be assigned the value of $c.C$. Because the access pattern is regular and the delivery delay is constant, the client will produce the same accesses, thus after at most two rounds, the client part of P holds for the current state.

Consider an arbitrary server s and let $s.C$ hold an arbitrary value. From action 4 of the server side algorithm it is clear, that $s.C$ will be reset at the end of every round. As argued above, the nodes will produce a regular access pattern within two rounds. This means that — starting from the next round — the values counted in $s.C$ (action 1) will from then on be the same in every subsequent round. Thus, the server part of P holds. So after at most two rounds P holds. □

Note that the proof does not consider the receipt of "move" instructions. Of course, if such instructions arrive during a particular round of the algorithm then the system will not stabilize within two rounds. This is why we have assumed above that the load balancing module evaluates its stability predicate (and sends "move" instructions) in intervals that are at least as long as two rounds. With this assumption Lemma 2 guarantees stabilization in every two successive rounds in which the stability predicate is not evaluated. (Recall that we assume globally synchronized clocks so triggering stability predicate evaluation on different nodes at the same time is not a problem.)

From Lemmas 1 and 2 the self-stabilization property of the algorithm can be followed under the given assumptions, which concludes the proof.

4.2. Dropping the Synchrony Requirement

In the previous course of this article we have assumed that the clocks of the individual processes are perfectly synchronized. It is a well known fact, that in real distributed systems this is impossible to achieve. However, using protocols like NTP [17], the clocks within the Internet for example can be synchronized with the bound of milliseconds [16]. In this section, we will discuss the consequences of such sources of asynchrony and show that they merely influence the efficiency of the scheme, not its correctness.

Now assume that the maximum clock offset between any two nodes within the system is δ. Then Figure 6 visualizes a possible scenario in which imperfect clock synchronization leads to a varying load count at a server. A client generates two accesses within phase i, but they arrive at the server within phase $i-1$ and $i+1$, respectively. This example shows that client accesses generated within δ time units around a phase switch may fall randomly into either of the two server phases. Thus, server access statistics become unstable, i.e., they may vary from phase to phase, even if the client access pattern has become regular.

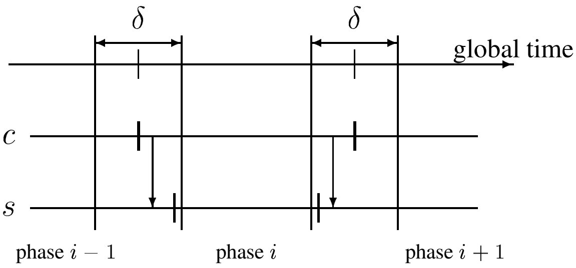

Figure 6. Possible scenario if maximum clock offset is δ.

However, it should be clear that the amount of possible accesses which can be the cause of such instability is proportional to δ. If δ is small, especially when compared to the phase length θ, then the amount of "variable load" will decrease compared to the total load of a server. In practice, a load balancing scheme will not try to level loads perfectly; it will let load vary between some threshold values before taking action. Hence, although clocks may not be perfectly synchronized, stability can nevertheless be reached if they are synchronized "sufficiently well". In general this means that divergence of clocks is a source of instability of the system, but this instability is bounded by the asynchrony of local clocks.

4.3. Performance Considerations

Abstract Analysis. Assuming that the distribution modules reside within the client side and server side proxies, this service may be introduced without additional messages during normal operation. The number of "move" instruction messages distributed by the load balancing module depends on the particular load balancing scheme used. Whenever the stability predicate is evaluated, at most one message per client per server is generated. (As discussed above, the time it takes to evaluate the stability predicate should be not shorter than two phases).

The data structures necessary to implement the scheme use simple counters and are bounded by the number of clients (for the server side distribution module) and the number of servers (for the client side distribution module). As the set of clients that may access a particular server during any execution may be far less than the total number of clients, the data structures on the server may be reduced to the size of this set, which is a substantial improvement.

Applicability. A question concerning the applicability of our scheme is justified at this point: ordinary clients are neither the sole source of high server loads, nor is their access pattern regular; why should this method be used at all? First of all, we recommend positioning the client side distribution module within a client side proxy. This proxy may serve many applications and thus also many users. Therefore a high access rate from an individual client is not negligible, especially if the server is inadequately equipped with computing resources. The second objection concerning the irregularity of client access patterns can be argued against by stating that clients merely have to reproduce the same *number* of accesses within a phase if clocks are sufficiently synchronized. Overall, we find that the low additional message and space complexity introduced by our scheme result in a good cost-benefit ratio.

5. Related Research

There exists a diverse variety of load balancing algorithms from many areas of computer science in the literature (see for example the papers by Arora and Gouda [5] or Arndt et al. [2]). Using the terminology from distributed systems of Shivaratri et al. [19], we have concentrated on *dynamic load distributing algorithms* where the *load index* is defined as the number of accesses per phase. Since every access can be viewed as an individual "task", we deal with *non-preemptive* load distribution.

We have surveyed the most recent literature on schemes to distribute load among replicated servers. Such schemes fall mainly into two categories: In *client oriented* schemes, the client successively collects statistics about the current performance of the servers it contacts. These statistics are used to predict the response time of the servers and thus make a good choice of which server to select [10, 11]. In *server oriented* approaches a dedicated machine (usually a switch or router) assigns incoming requests to a cluster of servers according to some local selection scheme (usually round robin) [1, 9]. While the latter approach — due to the single common entry point — is neither scalable nor fault-tolerant the former suffers from the need to regularly update aging statistics.

We are aware of three methods [7, 8, 15] that — like our method — use client/server cooperation to level loads on replicated servers. They all address two separate concerns [8]: (1) the problem of disseminating data closer to clients, and (2) finding the nearest replica.

These two concerns are tackled by a cooperative replication strategy based on client access statistics collected by the servers. When requesting a document, a client queries the primary server for this document by accessing a given replica. The primary server then redirects the request to a "nearer" replica or actively pushes the document there if such a replica does not currently exist [8, 15]. Clients keep track of these redirections and thus eventually make better server choices. This can be done in a hierarchical [7] and transparent manner by augmenting client and server side proxies [7, 8, 15] and using standard protocols like HTTP [7].

These three schemes all use client/server cooperation for load distribution and thus may seem similar to the method presented here. However, they are mainly concerned with improving document retrieval times by reducing network congestion. Load balancing on servers is a side effect of these schemes as they assume that accesses to near replicas will distribute the load evenly. However, a client always has a single server to contact for a replicated item at any point in time and thus these methods fail when a single client is the sole reason for high server load.

Additionally, dynamic load balancing is an excellent topic for self-stabilization because solutions to the problem must be adaptive and can naturally exploit the closure and convergence features of stabilization (see for example the papers by Arora and Gouda [4, 5]). Interestingly, none of the schemes cited above addresses the subject within the context of self-stabilization although the self-stabilization property implies non-masking fault tolerance against all forms of transient faults.

Another point which distinguishes our approach from the ones discussed above is its modularity. Note that we have not proposed yet another novel load distribution algorithm but rather an extension to existing schemes that have some specific interface. Such a scheme can be plugged into the "load balancing module" as we have called it. The different policies which Shivaratri et al. [19] use to classify load distribution schemes (*transfer policy*, *selection policy*, *location policy* and *information policy*) are all determined by this module. While there may exist algorithms which have similar overall functionality to our composed system, we are unaware of any approach which possesses the distinctive modularity features of our method.

6. Conclusions and Future Work

In this paper, we have presented a simple method to provide load balancing for replicated servers on a per-access basis. To the authors' knowledge this is the first approach to extend the many-to-one mapping between clients and servers to a full-fledged many-to-many mapping. Our scheme is modular in the sense that it can be used to enhance existing load balancing schemes and it does not interfere with replica consistency algorithms. The necessary prerequisites are sufficiently well synchronized clocks on the participating nodes, which is no undue restriction since current protocols such as NTP provide sufficient clock synchrony and the quality of synchronization only influences efficiency and not correctness. Given the moderate additional costs we find that our scheme is well-suited to advance the quality of network operation in replicated environments, thus combining coarse-grained load balancing through existing replication schemes and fine-grained load balancing through our method.

Future work will comprise pursuing (among others) the following question: When nodes are instructed to "move" a certain amount of accesses to a different server it is desirable that the benefits of load migration are not annihilated by additional communication costs of accessing far away servers; so how can additional heuristics be added to our scheme to prevent costly load migration within the *split* function? Finally, we plan to implement this scheme within our existing framework for transparent document replication for the World-Wide Web [21].

Overall we find that our scheme is an interesting example of a self-stabilizing algorithm [12, 18] which exemplifies that self-stabilization is more than "merely" a method of fault tolerance [3]; the notion of stability is so central that it seems to encapsulate a general distributed programming paradigm [6].

Acknowledgments

We wish to thank the anonymous referees for their instructive comments.

References

[1] Load-balancing internet servers. IBM Redbook, Document Number SG24-4993-00, June 1998. Available at http://www.redbooks.ibm.com/SG244993/4993fm.htm.

[2] O. Arndt, B. Freisleben, T. Kielmann, and F. Thilo. Dynamic load distribution with the WINNER system. In *Proc. of the Workshop "Anwendungsbezogene Lastverteilung" (ALV'98)*, pages 77–88, München, Germany, 1998. Technische Universität München.

[3] A. Arora and M. Gouda. Closure and convergence: a foundation of fault-tolerant computing. *IEEE Trans. on Softw. Eng.*, 19(11):1015–1027, 1993.

[4] A. Arora and M. Gouda. Load balancing: an exercise in constrained convergence. In *Proc. of the 9th Int. Workshop on Distr. Alg. (WDAG95)*, pages 183–197, 1995.

[5] A. Arora and M. G. Gouda. On the correctness criteria of load balancing programs. Internet: ftp://ftp.cis.ohio-state.edu/pub/anish/papers/load-balancing.ps.gz, Apr. 1997.

[6] A. Arora, M. G. Gouda, and G. Varghese. Constraint satisfaction as a basis for designing nonmasking fault-tolerant systems. *Journal of High Speed Networks*, 5(3):293–306, 1996.

[7] M. Baentsch, G. Molter, and P. Sturm. Introducing application-level replication and naming into today's web. *Computer Networks and ISDN Systems*, 28(7–11):921–930, May 1996.

[8] A. Bestavros. Demand-based document dissemination for the world-wide web. Technical Report 95-003, Boston University, Computer Science Department, Feb. 1995.

[9] T. P. Brisco. DNS support for load balancing. Internet Request for Comments 1794, Apr. 1995.

[10] R. Carter and M. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report 96-007, Boston University, 1996.

[11] M. Crovella and R. Carter. Dynamic server selection in the internet. Technical Report 95-014, Boston University, 1995.

[12] E. W. Dijkstra. Self stabilizing systems in spite of distributed control. *Comm. of the ACM*, 17(11):643–644, 1974.

[13] R. A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California, Santa Cruz, Dec. 1992. Also published as Technical Report UCSC-CRL-92-52.

[14] P. Grønning, T. Q. Nielsen, and H. H. Løvengreen. Stepwise development of a distributed load balancing algorithms. In *Proc. of the 4th Int. Workshop on Distr. Alg. (WDAG90)*, pages 151–168, 1990.

[15] J. S. Gwertzman and M. Seltzer. The case for geographical push-caching. In *Proc. of the Fifth Annual Workshop on Hot Operating Systems HotOS'95, Orcas Island, WA*, pages 51–55, May 1995.

[16] D. L. Mills. Measured performance of the network time protocol in the internet system. Internet Request for Comments RFC 1128, Oct. 1989.

[17] D. L. Mills. Network time protocol (version 3). Internet Request for Comments RFC 1305, Mar. 1992.

[18] M. Schneider. Self-stabilization. *ACM Comp. Surveys*, 25(1):45–67, 1993.

[19] N. G. Shivaratri, P. Krueger, and M. Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–45, 1992.

[20] G. Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[21] O. Theel, H. Pagnia, A. Berger, and A. Liebig. Enhancing the World-Wide Web through a transparent integration of document replication. In *Proc. of the 2nd World Multiconf. on Systemics, Cybernetics and Informatics, Orlando, Florida*, Vol. 3, pages 334–340, July 1998.